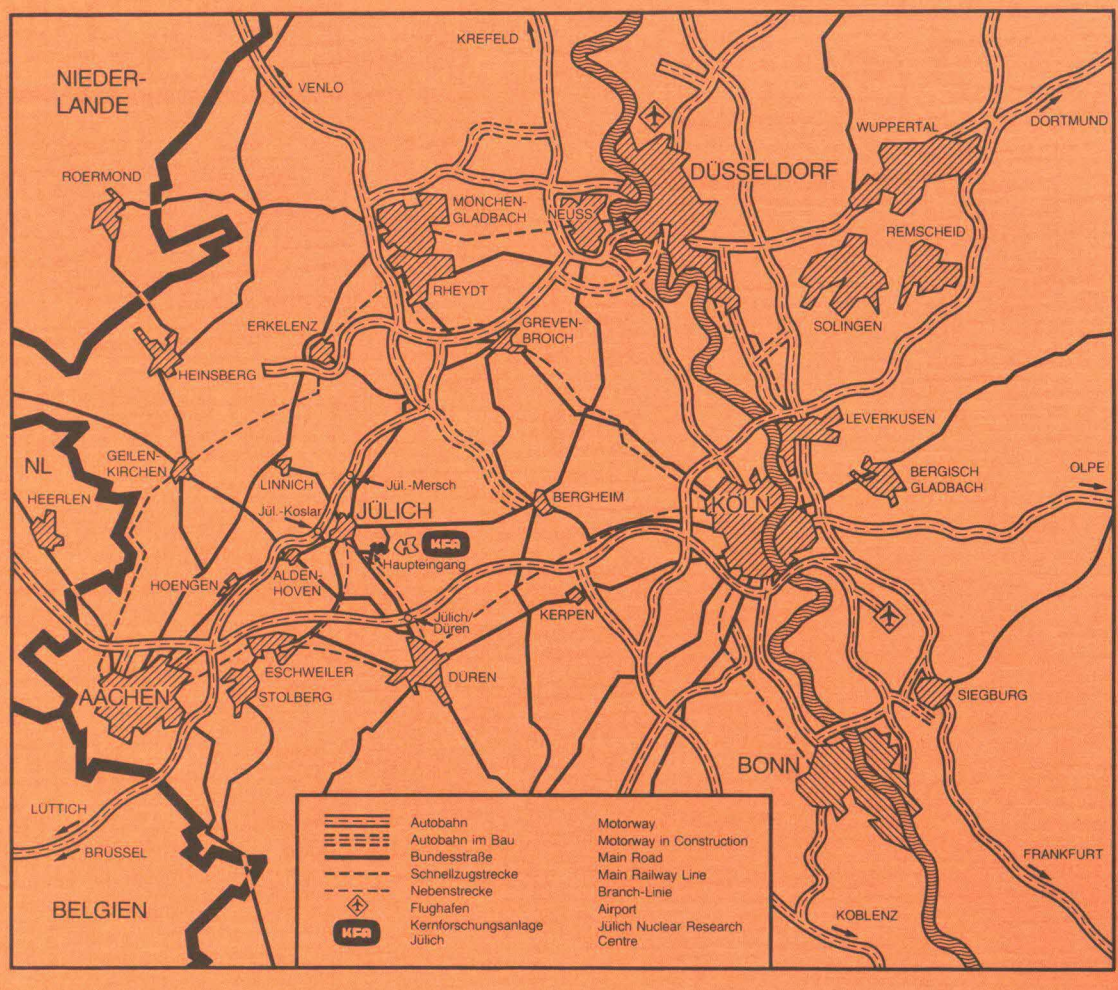
KERNFORSCHUNGSANLAGE JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik

Suchalgorithmen für Zeichenketten auf SIMD-Maschinen

von

J. Tappe

Jül-Spez-302
Februar 1985
ISSN 0343-7639



Als Manuskript gedruckt

Spezielle Berichte der Kernforschungsanlage Jülich – Nr. 302
Zentralinstitut für Angewandte Mathematik Jül-Spez-302

Zu beziehen durch: ZENTRALBIBLIOTHEK der Kernforschungsanlage Jülich GmbH
Postfach 1913 · D-5170 Jülich (Bundesrepublik Deutschland)
Telefon: 02461/610 · Telex: 833556-0 kf d

Suchalgorithmen für Zeichenketten auf SIMD-Maschinen

von

J. Tappe*

*Lehrstuhl B für Mathematik der RWTH Aachen

Suchalgorithmen für Zeichenketten auf SIMD-Maschinen

Jürgen Tappe

1. Einleitung

In der vorliegenden Arbeit behandeln wir das Problem, eine oder mehrere (kurze) Zeichenketten PAT_i, i = 1, 2, ..., n (patterns) in einer (langen) Zeichenkette ST (string) zu finden. Die einfachste Suchmethode besteht darin, für alle j die Zeichen ST(j), ST(j+1), ... mit den Zeichen $PAT_i(1)$, $PAT_i(2)$, ... so lange zu vergleichen, bis keine Übereinstimmung mehr vorliegt oder PAT_i gefunden wurde. Bezeichnen wir mit l die Länge von ST, mit l_i die Länge von PAT_i, so ist im ungünstigsten Falle der Zeitbedarf des Suchvorgangs von der Ordnung

$$l \cdot \sum_{j=1}^{n} l_j \cdot$$

Dieser Zeitbedarf wird z.B. erreicht bei Mustern der Form aaa...ab in Ketten der Form aaaa...a. Für realistische Daten aus der Textverarbeitung ist jedoch der Zeitbedarf linear.

Um die Einsatzmöglichkeit von SIMD-Maschinen auf das obige Problem zu erörtern, betrachten wir zunächst in § 2 die bekannten schnellen seriellen Suchverfahren aus [1],[2] und [3]. In § 3 folgt eine Parallelisierung des obigen elementeren Verfahrens, die sich für Parallelprozessoren mit Prozessorelementen einfacher Architektur eignet. Das schnellste serielle Verfahren aus [3], das allerdings nur für die Suche nach einem Zeichenmuster geeignet ist, wird in § 4 auf eine Architektur vom Typ CYBER 205 übertragen. In § 5 betrachten wir eine Pipelineversion des Algorithmus aus [3]. Um eine zufriedenstellende Synchronisation zu erreichen, gehen wir von kurzen Suchworten in großen Alphabeten aus, was in der Praxis durch Zusammenfassen mehrerer Zeichen zu einem erreicht wird. Dies kommt insbesondere wortorientierten Maschinen entgegen. Bei simultaner Anwendung des Algorithmus für mehrere Muster treten Suchprobleme auf, die allerdings durch eineindeutige Hashadressierungen gelöst werden können. Eine Analyse dieser Suchprobleme ist in § 6 gegeben. Beschränken wir uns bei der Mustererkennung auf ein einzelnes Suchwort, so treten die o.g. Schwierigkeiten nicht auf. In diesem Falle scheint eine Architektur vom Typ CRAY X-MP für einen flexiblen

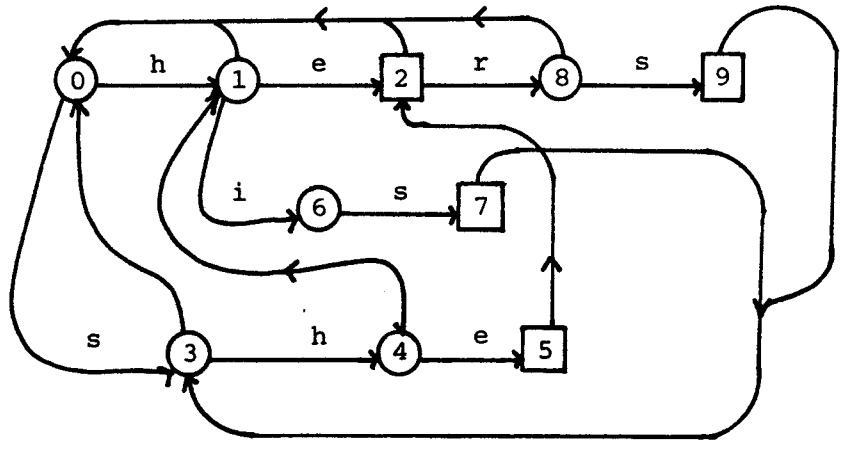
Einsatz besonders geeignet; u.a. liegt dies daran, daß für nur ein Suchwort auf eine der sonstigen Rechengeschwindigkeit angepaßte Gather-Funktion verzichtet werden kann. Eine in FORTRAN und CAL implementierte Routine ist in § 7 angegeben.

Dieser Beitrag entstand in Zusammenarbeit mit dem Zentralinstitut für Angewandte Mathematik der Kernforschungsanlage Jülich GmbH. Das dort vorhandene Interesse an Algorithmen, die für das Suchen von Zeichenketten auf Vektorrechnern wie der CRAY X-MP geeignet sind, veranlaßten mich zur Beschäftigung mit dieser Problemstellung. Für die Unterstützung bei der Implementierung möchte ich Herrn Groten herzlich danken.

2. Schnelle serielle Methoden

In diesem Abschnitt betrachten wir ohne eine Vertiefung der Details die bekannten schnellen seriellen Methoden. Wir betrachten zunächst das Verfahren aus [1], das geeignet ist, in ST mehrere Suchworte zu finden. Schränkt man es auf ein Suchwort ein, so ergibt sich der Algorithmus aus [2], auf den wir hier nicht mehr näher eingehen.

Die Suche in [1] erfolgt anhand eines Graphen für dessen Erstellung wir auf die Originalarbeit verweisen. Für die Suchworte "he, she, his, hers" ist er wie folgt gegeben:



Ausgabe:

2	he
5	he, she
7	his
9	hers

Gewisse, im obigen Beispiel die nach rechts weisenden Pfeile des Graphen sind durch Buchstaben gekennzeichnet. Zudem beginnt in jeder Ecke ein nicht gekennzeichneter Pfeil. Im Startzustand ist ST(1) das aktuelle Zeichen und 0 die aktuelle Ecke. Es sei nun i die aktuelle Ecke und ST(j) das aktuelle Zeichen. Stimmt ST(j) mit der Kennzeichnung eines Pfeiles mit Ursprung in i überein, so wird der Endpunkt des Pfeils zur aktuellen Ecke, ST(j+1) das aktuelle Zeichen.

Ist ST(j) von allen o.g. Kennzeichnungen verschieden, so ist der Endpunkt des Pfeils aus i ohne Kennzeichnung aktuell, und ST(j) bleibt das aktuelle Zeichen. Bei gewissen Ecken erfolgt eine Ausgabe. Bei diesem Verfahren besteht das Hauptproblem darin, für jede Ecke den Test, ob ein aktuelles Zeichen mit einer der zugehörigen Kennzeichnungen übereinstimmt, zu organisieren. Die einfachste Methode besteht darin, für jede Ecke ein Feld zu speichern, dessen Dimension mit der Größe des Alphabets übereinstimmt, wodurch aber der Speicheraufwand groß werden kann.

Schneller als das obige Verfahren, jedoch beschränkt auf ein Suchwort ist der folgende Algorithmus aus [3]: Es seien ST und PAT wie oben gegeben, und mit m bezeichnen wir die Länge von PAT. Für die Zeichen z des Alphabets sei

$$d_1(z) = \begin{cases} m, \text{ falls } z \notin PAT \\ m-j, \text{ falls } PAT(j) = z,\ PAT(k) \neq z \text{ für alle } k > j. \end{cases}$$

Ferner definiert man für die $j \in \{1, 2, \ldots, m\}$ die Zahl a(j) als die kleinste natürliche Zahl a für die PAT(i) = PAT(i-a) für i = j+1, ..., m und PAT(j-a) ≠ PAT(j) gilt; dabei ist vereinbart, daß die obigen Bedingungen immer erfüllt sind, für die Argumente kleiner oder gleich 0 auftreten. Damit ist

$$d_2(j) := a(j) + m - j$$

definiert. Wir starten den Algorithmus, indem wir einem Zeiger i den Wert m zuweisen. Wir vergleichen dann die Zeichen PAT(m) und ST(i), PAT(m-1) und ST(i-1) usw., bis in Position j zum ersten Male PAT(j) und ST(i-m+j) voneinander verschieden sind. Daraufhin weisen wir i den Wert $i+\max(d_1(j), d_2(ST(i-m+j)))$ zu und fahren fort wie oben. Im günstigsten Falle, bei dem alle Zeichen ST(k•m), k = 1, 2, ... nicht in PAT vorkommen, zeigt das obige Verfahren nach nur l/m Schritten (l = Länge von ST) an, daß PAT nicht in ST vorkommt.

3. Elementare parallele Mustererkennung

In diesem Abschnitt setzen wir voraus, daß eine SIMD-Maschine vorliegt, die n Prozessoren hat (z.B. ICL-DAP). Jeder Prozessor verfüge über einen gewissen lokalen Speicher. Gegeben sei eine Zeichenkette ST und ein Suchwort PAT der Länge m. Mit PE_1, PE_2, ... , PE_n bezeichnen wir die einzelnen Prozessoren.

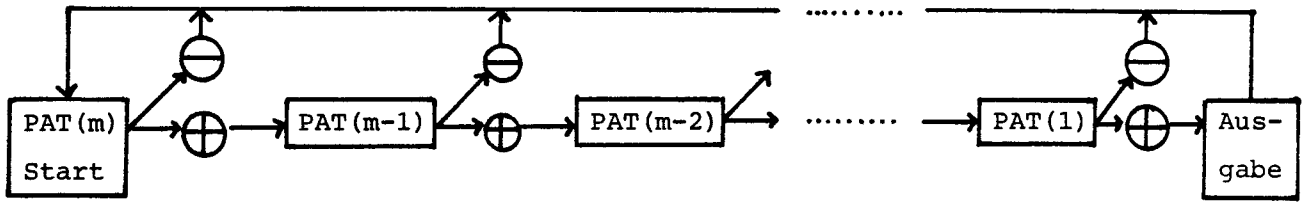
Zunächst gehen wir davon aus, daß im Speicher von PE_i die Zeichen ST((i-1)m+1), ..., ST((i+1)m-1) vorliegen. Je nach der Beschaffenheit der Verbindung zwischen dem Hauptspeicher und den lokalen Speichern ist die obige Anordnung in zwei Schritten möglich. Nach m^2 Vergleichen ist dann das Vorkommen von PAT in der Teilkette ST(1), ..., ST((n+1)m-1) überprüft. Die Ausgabe erfolgt zweckmäßigerweise über eine master unit. Pro Zeiteinheit (bezogen auf die Zeit für einen Vergleich von Zeichen) werden im Mittel bis zu n/m Positionen von ST auf Übereinstimmung mit PAT überprüft. Da m in der Regel klein ist, ist beim obigen Verfahren durchaus eine hohe Leistung möglich. Dennoch sind gewisse Nachteile nicht zu übersehen. Die Speicherorganisation hängt von m ab, wodurch gewisse Probleme auftreten, wenn das Verfahren unter Beibehaltung von ST auf mehrere Muster angewendet werden soll. Ebenso reduziert sich die Leistungsrate n/m wenn die Länge von ST kleiner als m•n ist. Eine Aufteilung von ST in kleinere Abschnitte als 2m-1 wirkt diesem Effekt entgegen, bewirkt aber Zugriffskonflikte beim Laden der lokalen Speicher.

Das folgende Verfahren ist in der Grenze weniger effektiv, dafür aber flexibler als das obige. Es wird dabei aber vorausgesetzt, daß simultan Daten von PE_i zu PE_{i+k}, $k \in Z$ weitergegeben werden können. Wir übergeben nun die Elemente ST(1), ..., ST(n) (je eins) an die Prozessorelemente PE_1, ..., PE_n, ferner sei PE_i das Zeichen PAT(j) gegeben, sofern $i \equiv j \bmod(m)$. Ein paralleler Vergleich von ST(i) mit PAT(j) in PE_i liefert einen Bitvektor der Länge n. Durch rekursives Doppeln enthalten nach $\log m$ Schritten die Elemente PE_i mit $i \equiv 1 \bmod(m)$ die Information darüber, ob in Position i von ST eine Teilkette der Form PAT beginnt. Analog überprüft man die Positionen j mit $j \equiv 2, 3, .., m \bmod (m)$. Insgesamt sind daher für eine Kette ST der Länge n ca. $m(1 + \log m)$ Operationen auszuführen. Bei der genauen Analyse der Ausführungszeit spielt es dabei eine große Rolle wie schnell ein Bit von PE_i an PE_{i+k} übertragen werden kann. Operieren die Prozessoren (wie beim DAP) nur auf einzelnen Bits, so sind die logischen Operationen beim rekursiven Doppeln kürzer als die Vergleichsoperationen (wenn z.B. ein Alphabet zugrundeliegt, das pro Zeichen 8 Bits benötigt). Bezogen auf die Ausführungszeit eines Vergleichs von Zeichen kann dann der Zeitbedarf niedriger als $m(1 + \log m)$ angesetzt werden.

4. Zur Parallelisierung des Boyer-Moore-Algorithmus

Zur Parallelisierung von string-searching-Algorithmen sei zunächst folgendes bemerkt. Prinzipiell besteht die Möglichkeit, ähnlich wie in Abschnitt 3, die Zeichenkette ST auf verschiedene Prozessoren zu verteilen und jeden Prozessor

nach einem geeigneten Verfahren suchen zu lassen. Dies ist allerdings uneingeschränkt nur auf MIMD-Rechnern möglich. Für SIMD-Architekturen ist eine Synchronisation nötig. In § 3 wurde dies erreicht auf Kosten eines quadratischen Aufwands pro Prozessor. Wir versuchen nun, am Beispiel des Boyer-Moore-Verfahrens (vgl. [3]) diesen Aufwand zu reduzieren. Dazu macht man sich klar, daß der logische Ablauf dieses Algorithmus durch folgendes Diagramm dargestellt werden kann:



Es wird dabei ein Zeichen PAT(i) mit einem Zeichen ST(j) verglichen. Stimmen sie überein, so erfolgt der Übergang von i und j zu i-1 und j-1 bzw. im Falle i = 1 erfolgt eine Ausgabe. Andernfalls wird i durch m ersetzt und das Inkrement für j anhand von d_1 und d_2 (vgl. § 2) bestimmt. Auf einem SIMD-Rechner ist es daher denkbar, nach einem parallelen Vergleichsschritt die Prozessorelemente mit einer logischen Maske in solche mit positivem und negativem Ergebnis zu trennen und für beide Gruppen die nötigen Folgeschritte separat auszuführen. Wir wollen nun zeigen, wie man das Suchen mit k Prozessoren auf einer Pipelinemaschine mit einer der CYBER 205 ähnlichen Architektur simuliert:

Es sei ST der (lange) Textvektor in dem das Vorkommen des (kurzen) Vektors PAT überprüft werden soll. Wir benötigen nun mehrere Vektoren der Dimension k: PI(j) ist der Index des aktuellen Zeichens von PAT im j-ten Prozessor, PK(j) das zugehörige Zeichen, d.h. PAT(PI(j)) = PK(j); analog sind SI und SK bezogen auf ST (anstelle von PAT) definiert. Der Startwert für alle PI(j) beträgt m (Länge von PAT), die Startwerte von SI hängen davon ab, wie der Textvektor ST auf die "Prozessoren" verteilt wird (Simulation lokaler Speicher). Zudem seien die in § 2 verschriebenen Funktionen d_1 und d_2 in Form von Vektoren gegeben.

Ablauf der Rechenschritte:

1. Der parallele Vergleich von PK und SK liefert eine Bitmaske M.

2. Die Vektoren PI und SI komprimiere man anhand von M und seinem Komplement $\underline{M}$, wodurch die Vektoren PI1, PI2, SI1, SI2 entstehen, Ferner entstehe SK2 durch Komprimieren mit $\underline{M}$ aus SK.

3. Durch parallele Subtraktion reduziere man die Komponenten von PI1 und SI1 um 1.

4. Ein paralleler Vergleich der Komponenten von PI1 mit 0 liefert eine Bitmaske N.

5. Jede 1 in N entspricht einer Teilkette von ST, die mit PAT übereinstimmt, es erfolgt die Ausgabe:

   Komprimiere SI1 mit N, woraus die Startpositionen der Teilketten gegeben sind.

6. Ersetze die Komponenten von PI1, die zu den Einsen von N gehören durch m (realisierbar durch einen Komprimier- und einen Mischbefehl).

7. Erhöhe die Komponenten von SI1, die zu den Einsen von N gehören um m (realisierbar durch zwei Komprimier-, einen Addier- und einen Mischbefehl).

8. Bestimme einen Vektor D1 mit "Gather" aus d_1 anhand von SK2, den Vektor D2 aus d_2 anhand von PI2.

9. Man berechne den Vektor D dessen Komponenten die Maxima der Komponenten von D1 und D2 enthalten.

10. Addiere die Vektoren SI2 und D, Ergebnis in SI2.

11. Anhand von M mische man PI1 mit einem Vektor, dessen Komponenten sämtlich gleich m sind, das Resultat ergibt PI.

12. Die aktualisierte Version von SI entsteht durch Mischen von SI1 und SI2 anhand von M.

13. Update von PK: "Gathering" in PAT anhand von PI
    Update von ST: "Gathering" in ST anhand von SI

14. Zur Speicherkontrolle müssen die Komponenten von SI geprüft werden. Dabei entsteht eine Bitmaske anhand derer die "Prozessorelemente" mit "Speicherüberschreitung" durch Komprimieren von PK, PI, SK, SI abgeschaltet werden.

15. Falls noch Prozessoren vorhanden: Goto 1.

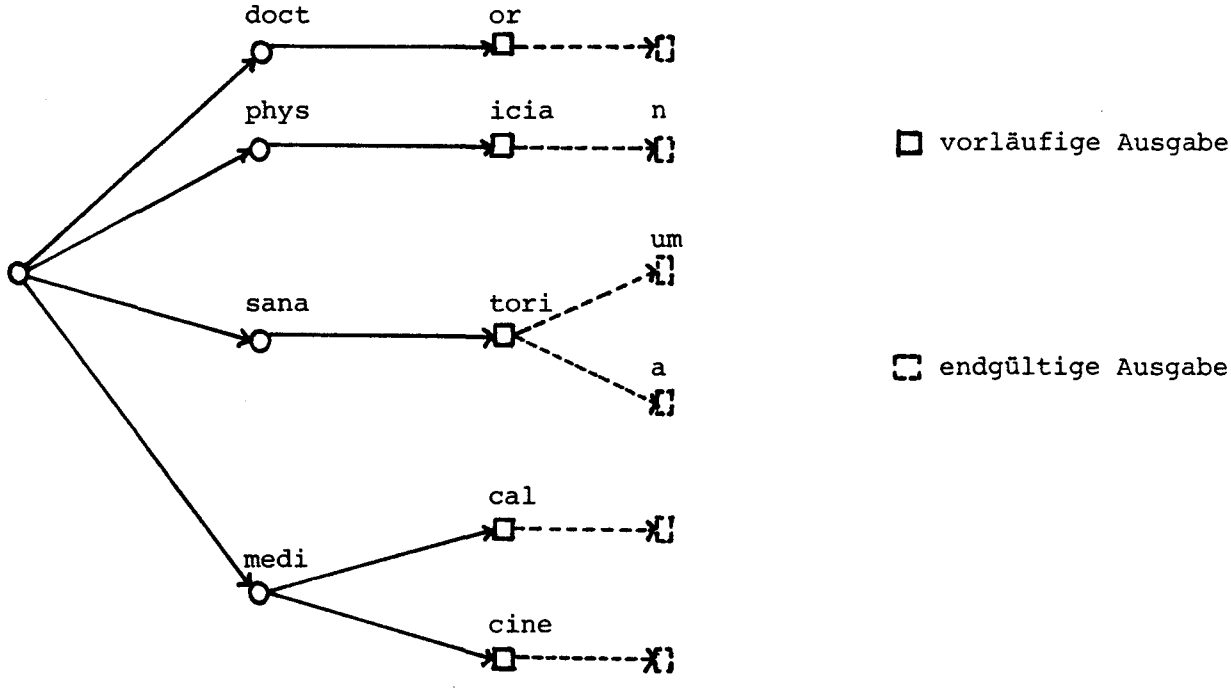
Die Vorteile des obigen Algorithmus liegen darin, daß viele Operationen parallel, d.h. durch Pipelining bearbeitet werden, was sich beschleunigend auswirkt. Es muß allerdings auch auf gewisse Mängel bzw. Schwierigkeiten hingewiesen werden. Die Aufspaltung in "Prozessoren" mit positiven und negativen Vergleichsergebnis kostet aus zweierlei Gründen Zeit. Zum einen ist stets ein Teil der Prozessoren inaktiv. Ferner erfordert die Aufspaltung zusätzliche Komprimier- und Mischoperationen, die ca. 30 - 40 % des gesamten Aufwandes beanspruchen. Größere Probleme ergeben sich jedoch bei der Organisation der Daten. So können z.B. auf der CYBER 205 nur Vektoren (und damit Zeichenketten ST) der Länge 65535 eingegeben werden. Bei nur 100 (simulierten) Prozessoren steht jedem Prozessor ein lokaler Speicher für 655 Zeichen auf ST zur Verfügung, der zu klein erscheint.

Wünschenswert für den parallelen Boyer-Moore-Algorithmus wäre ein Parallelprozessor, dessen Prozessorelemente zwar durch einen einfachen Befehlsstrom gesteuert werden dürfen, die aber über hinreichend viel RAM verfügen. Eine Implementierung auf einer Maschine mit CRAY-ähnlicher Architektur erscheint schwierig, wenn nicht unmöglich, da eine ausreichend schnelle Gather-Funktion (für den random access) und eine Komprimier-Funktion zur Erzeugung konstanter Inkremente fehlen.

5. Parallelisierung des Aho-Corasick-Algorithmus

In diesem Abschnitt wollen wir uns neben der Parallelisierung des o.g. Verfahrens mit zwei weiteren Gesichtspunkten beschäftigen. Zum einen wollen wir uns bei den Suchworten auf Zeichenketten von höchstens acht Zeichen beschänken. Dies reicht zwar bei der Suche z.B. nach Worten in englischer Sprache in der Regel nicht aus, reduziert aber den Aufwand derart, daß eine nachfolgende serielle Überprüfung der fehlenden Zeichen kaum ins Gewicht fällt. Ferner wollen wir in Betracht ziehen, daß viele Maschinen wortorientiert sind (z.B. mit 32 Bit pro Wort). In der Textverarbeitung wird in der Regel ein Zeichen mit 8 Bit belegt, so daß pro Wort vier Zeichen gespeichert werden können. Bei Algorithmen, die auf dem Vergleich einzelner Zeichen beruhen, treten also pack- und unpack-Probleme auf. Wenn wir nun davon ausgehen, daß sämtliche Suchworte in zwei aufeinander folgenden Speicherworten enthalten sind und eine Vergleichsoperation wegen der Wortorientiertheit der Maschine für ein bis vier Zeichen die gleiche Zeit beansprucht, so sollte diesem Aspekt Rechnung getragen werden.

Dies bedeutet ein Übergang zu einem Alphabet mit 2^{32} bzw. 2^{28} Buchstaben, je nachdem, welcher Code (EBCDIC oder ASC II) zugrundeliegt. Dabei reduzieren sich die Wortlängen der Suchworte durch die obige Voraussetzung auf maximal 2, während sich die Länge der Zeichenkette ST im wesentlichen nicht ändert, da alle Positionen bezogen auf das kleine Alphabet geprüft werden müssen. Der beim Algorithmus von Aho und Corasick angegebenen Graph (vgl. § 2) eignet sich aufgrund von Synchronisationsproblemen nicht für SIMD-Maschinen. Durch die oben eingeführten Einschränkungen ist es jedoch möglich, den Suchvorgang in zwei Stufen ablaufen zu lassen, d.h. die Steuerung erfolgt anhand eines gerichteten Baumes in dem alle Wege eine Länge haben, die kleiner oder gleich 2 ist. Für die Suchworte "doctor, physician, sanatorium, sanatoria, medical, medicine" hat dieser folgende Form:



Die gestrichelten Ecken und Pfeile deuten die serielle Nachbereitung an. Es ist klar, daß bei dieser Vorgehensweise dem Vergleich der ersten vier Zeichen die größte Bedeutung zukommt. Liegt nur ein Suchwort vor, so treten keine Probleme auf. Ein linearer Suchvorgang im Falle mehrerer Suchworte ist zu vermeiden, da dies bei der vorliegenden Situation nahezu dem mehrfachen Anwenden des Algorithmus für nur ein Suchwert gleichkommt und damit die Anzahl der Suchworte in die Komplexität des Algorithmus eingeht. Ein Feld, das mit den Buchstaben des hier angenommenen Alphabets indiziert ist, kommt wegen der großen Zahl von 2^{28} bzw. 2^{32} Zeichen nicht in Frage. Ist die Zahl der Suchworte (mit ca. 60) nicht allzu groß, so ist eine Hashadressierung möglich, wie dies in § 6 beschrieben ist. Ein weiteres Problem stellt die Verzweigung von der ersten zur zweiten Stufe des Suchvorgangs dar. Auch hier werden aus den "Endstücken" der Suchworte

Hashadressen ermittelt, die in Kombination mit den in der ersten Stufe ermittelten Adressen eine einwandfreie Steuerung gewährleisten. Wir betrachten nun eine Version des Suchalgorithmen für eine CYBER 205-Architektur.

Voraussetzungen

1. Die Zahl der Suchworte ist kleiner oder gleich 64.

2. Die Zeichenkette ST sei in Form eines Vektors V gegeben, wobei V(i) die Zeichen ST((i-1)•4+1), ..., ST(i•4) enthält. Die Suchworte sind linksbündig, der späteren Maskierung angepaßt, in zwei Worten $W_i(1)$, $W_i(2)$ gespeichert.

3. Gemäß § 6 seien zwei ganze Zahlen p_1, p_2 gegeben, so daß für alle $i \neq i', j$ gilt: $W_i(i) \neq W_{i'}(j) \bmod(p_j)$, sofern $W_i(j) \neq W_{i'}(j)$, wobei wir die $W_i(j)$ als ganze Zahlen interpretieren. Die Reste einer ganzen Zahl k bei Division durch p_j wollen wir $R_j(k)$ nennen.

4. Es sei A eine Liste mit den Inhalten $W_i(1)$, A(0) enthalte ein "verbotenes Symbol".

5. Es sei S ein ganzzahliges Feld der Dimension P_1 mit folgenden Eigenschaften: $A(S(R_1(W_i(1)))) = W_i(1)$, für $x \neq R_1(W_i(1))$ sei $S(x) = 0$.

6. Es sei T ein ganzzahliges Feld der Dimension P_2 und E ein Feld der Dimension 4096 mit $E(T(R_2(W_i(2))) * 2^6 + S(R_1(W_i(1)))) = W_i(2)$, für $x \neq R_2(W_i(2))$ sei $T(x) = 0$, und E(y), enthalte ein verbotenes Symbol für $y < 64$.

7. Für Ausgabezwecke sei P ein Vektor mit P(i) = i.

Algorithmus

Die Zahl der (simulierten) Prozessoren beträgt ein Viertel der Länge von ST (d.i. die Dimension von V). Es werden in einem Schritt die Positionen ST(j+i 4), i = 1, 2, 3, ... als Startpositionen für die Suchworte überprüft, so daß nach vier Schritten die Suche beendet ist. Ein Suchschritt läuft wie folgt ab:

1. Die "Anfangsstücke" $W_i(1)$ der Suchworte werden auf ihre Längen geprüft, haben alle Suchworte mindestens 4 Zeichen, wie dies in der Praxis sicher oft vorkommt, so sind die Schritte 2 bis 15 mit V' = V zu durchlaufen. Kommen jedoch auch Worte mit 1, 2 oder 3 Zeichen vor, so ist für jede dieser vorkommenden Wortlängen i $\varepsilon\{1, 2, 3\}$ eine maskierte Version von V in V' abzulegen. Entsprechend oft sind die Schritte 2 bis 5 zu durchlaufen.

2. Berechne die Hashadressen $R_1(V'(j))$.

3. Man bestimme einen Indexvektor I mit "Gather" aus dem Vektor S anhand der Adressen aus 2.

4. Der Vektor V'' entstehe mit "Gather" aus A anhand von I.

5. Der Vergleich von V' und V'' liefert einen Bitvektor C_i, i = aktuelle Maskenlänge. Für die Maskenlänge $i \leq 3$ erfolgt eine Ausgabe: Komprimiert man P mit C_i, so ergeben sich Startpositionen der Suchworte, komprimiert man I mit C_i, so sind damit die gefundenen Suchworte gekennzeichnet.

6. Abfrage: $C_4 = 0$ - Goto 15.

7. Der Vektor $\underline{V}$ entstehe aus V durch Komprimieren mit C_4 derart, daß genau die Komponenten V(j+1) vorliegen, für die V(j) bei Maskenlänge 4 (d.h. ohne Maskierung) eine Übereinstimmung brachte.

8. Komprimiere I mit C_4 zu $\underline{I}$.

9. Ähnlich wie oben sind die Schritte 9 bis 14 für bis zu 4 Maskenlängen auszuführen. Die jeweils aktuelle maskierte Version von $\underline{V}$ heiße $\underline{V}'$

10. Berechne die Hashadressen $R_2(\underline{V}'(j))$.

11. Berechne den Indexvektor J mit "Gather" anhand von 10 und den triadischen Ausdruck $Z = J * 2^6 + I$.

12. Bestimme V'' mit "Gather" aus E anhand von Z.

13. Der Vergleich von $\underline{V}'$ und $\underline{V}''$ liefert für die aktuelle Maskenlänge i den Bitvektor C_i'.

14. Das Komprimieren von P mit C_4 und C_i' und das Komprimieren von Z mit C_i' liefert die Daten für die Ausgabe der Ergebnisse.

15. Shift von V um 8 Bit.

Zum Abschluß dieses Paragraphen wollen wir uns mit dem Spezialfall beschäftigen, bei dem nur ein einzelnes Suchwort vorliegt. Diese Einschränkung wirkt sich an verschiedenen Stellen des Algorithmus zeitsparend aus. Die Hashadressierung ist dann überflüssig, mehrfache Durchläufe mit verschiedenen Maskenlängen entfallen. Zudem wird keine Gather-Funktion mehr benötigt. In dieser Situation erweist sich nun auch die Architektur der CRAY-Computer als zweckmäßig (Doppelshifts, Bildung von Vektormasken). Hinzu kommt noch, daß es nun problemblos möglich ist, beide Stufen des Algorithmus mit 64-Bit-Worten zu organisieren, so daß vom Suchwort bis zu 16 Zeichen geprüft werden, wodurch sich der Aufwand beim Nachbereiten reduziert. Das Problem im Algorithmus für mehrere Suchworte liegt darin, daß die Hashadressierung erst nach einer Maskierung möglich ist, diese aber von der Suchwortlänge abhängt. Dies erfordert unnötige, aber nicht zu vermeidende Mehrarbeit beim Maskieren. Eine Erweiterung des allgemeinen Algorithmus von 4 auf 8 Zeichen pro Algorithmenstufe ist daher nicht sinnvoll. Der Grund liegt darin: In vielen Fällen haben die Suchworte mehr als drei Buchstaben, so daß in der ersten Stufe die Maskierung entfällt. Zudem kann man davon ausgehen, daß die zweite Stufe nur wenig und auf kurzen Vektoren durchlaufen wird, so daß die durch das Maskieren nötige Schleife sich nicht allzu nachteilig auswirken dürfte.

Beim auf ein Suchwort eingeschränkten Algorithmus spart man durch die einfachere Organisation nicht unerheblichen Speicherplatz, zudem noch die Zeit für die Initiierung (Hashfunktion etc.).

Auch ohne vorliegende Vergleiche wird man davon ausgehen können, daß die Architektur der CYBER 205 besonders für lange Textketten im Hauptspeicher geeignet ist, wohingegen die häufige Anwendung des Verfahrens auf kürzere Zeichenketten (wenn z.B. nur kurze Abschnitte von ST vom Sekundärspeicher gelesen werden) der Architektur der CRAY-Maschinen entgegenkommt. Obgleich die o.g. Maschinen sicherlich nicht zur Textverarbeitung entworfen wurden, so zeigt es sich, daß ihre Eigenschaften durchaus bei der Mustererkennung in Zeichenketten eine Beschleunigung liefern können.

6. Eineindeutige Hashadressierung

In vielen Anwendungen tritt das Problem auf, ein gegebenes Datum in einer Datenmenge zu suchen. Dabei haben sich in vielen praktischen Fällen Hash-Verfahren bewährt, mit deren Hilfe die Zahl der Suchschritte reduziert werden. In § 5 entstand eine etwas andere Situation. Die zu durchsuchende Datenmenge, etwa die Menge der jeweils vier Anfangsbuchstaben der gegebenen Suchworte ist relativ klein. Dafür ist eine Adressierung gesucht, bei der die Suche in einem einzigen Schritt erfolgt. Da der 2^{28} bzw. 2^{32} Vierergruppen von Zeichen möglich sind, ist eine Adressierung mit Hilfe eines Feldes nicht möglich. Wir suchen daher eine Hashfunktion, die einerseits nur verhältnismäßig wenige Werte durchläuft, andererseits auf der gegebenen Menge von Zeichenketten paarweise verschiedene Werte liefert. Als mögliche Hashfunktion wählen wir, wie dies häufig üblich ist, die Restklassenbildung nach einer ganzen Zahl. Damit läßt sich das Problem wie folgt darstellen: Es seien natürliche Zahlen $a_1, a_2, \ldots, a_n$ gegeben. Gesucht ist eine natürliche Zahl m mit $a_i \not\equiv a_j \bmod(m)$ für alle $i \neq j$. Damit ist es möglich, einen Vektor A(o:m-1) zu definieren, der die Eigenschaft $A(\mathrm{Mod}(a_i,m)) = a_i$ hat. Damit kann man durch eine Restbildung, einen Zugriff auf A und einen Vergleich für eine beliebige Zahl b prüfen, ob sie in der Menge $a_1, a_2, \ldots a_n$ vorkommt. Die maximale Vektorlänge auf der CYBER 205 ist 65535, so daß wir uns auf $m \leq 65535$ beschränken wollen.

Eine Zahl m ist für unsere Adressierung genau dann geeignet, falls keine der Differenzen $a_i - a_j$ für $i \neq j$ durch m teilbar ist. Um eine bessere Übersicht zu behalten betrachten wir zunächst die Pimzahlpotenzen p^l mit $p^l \leq 65535$, aber $p^{l+1} > 65535$. Dies reduziert m auf ca. 5900 Werte (beachte $\pi(x) \sim x/\ln x$), die alle größer oder gleich 257 sind. Ein für uns ungünstiger Fall liegt also dann vor, wenn jede obige Potenz p^l in mindestens einer der $\frac{n(n-1)}{2}$ Differenzen $a_i - a_j$, $i > j$ vorkommt. Im schlimmsten Falle sind die Primzahlpotenzen disjunkt auf die Differenzen verteilt. Bei dem für § 5 relevanten Problem liegen die Differenzen (dem Betrage nach) unterhalb von 2^{32}. Man schätzt nun leicht ab, daß durch die Beschränkung und die relativ kontinuierliche Verteilung der Primzahlen jede Differenz im Mittel weniger als 3 Primzahlpotenzen p^l für sich beanspruchen kann. Daraus folgt, daß auf diese Weise eine eineindeutige Hashadressierung für mindestens 64 Anfangs- oder Endstücke von Suchworten möglich ist. Eine entsprechende Betrachtung liefert bei $m \leq 6500$ bei ca. 700 Primzahlpotenzen eine eineindeutige Hashadressierung für mindestens 20 Suchworte. Andere Zahlen, die keine Primzahlpotenzen sind, sind natürlich ebenso als Moduln m geeignet. Dabei ergibt sich nach dem chinesischen Restsatz, daß die Zerlegung der Zahlen $a_1, \ldots, a_n$ durch Restebildung nach m die gemeinsame Verfeinerung der entsprechenden Zerlegungen anhand der Primzahlpotenzfaktoren von m ist.

Eine Kombination mehrerer kleinerer Primzahlpotenzen kann durchaus ein vielversprechender Ansatz sein. Zudem sei noch darauf hingewiesen, daß die Zahl der Primzahlen, die kleiner oder gleich 2^{32} sind in der Größenordnung von 10^9 liegt, so daß eine Beschränkung bei der Zerlegung der Differenzen a_i-a_j auf die ersten 5900 bzw. 700 Primzahlen sehr unwahrscheinlich ist, d.h. i.a. sind günstigere Ergebnisse als 64 bzw. 20 Suchworte zu erwarten.

Wir beschließen diesen Abschnitt mit einigen Bemerkungen zur Bestimmung eines geeigneten Moduls m. Eine Möglichkeit besteht darin, wie oben bemerkt, die Zahl m als Produkt kleiner Primzahlpotenzen zu kombinieren. Will man eine vorgegebene Menge $m_1, m_2, \ldots, m_k$ bis zu einem erfolgreichen Test durchprobieren, so besteht einerseits die Möglichkeit, die einmal berechneten $\frac{n(n-1)}{2}$ Differenzen a_i-a_j, $i > j$ auf Teilbarkeit durch m_l, $l = 1, 2, 3, \ldots$ zu testen, oder aber die Werte $\mathrm{Mod}(a_j,m_l)$, $j = 1, 2, \ldots, n$ etwa anhand eines Sortiervorgangs auf paarweise Verschiedenheit zu überprüfen (zum Studium von Sortieralgorithmen vgl.[4]). Beide Methoden lassen sich auf Vektorrechnern zufriedenstellend realisieren.

7. Anhang

Im nachfolgenden FORTRAN-Programm PAT1/PAT2 für die CRAY X-MP werden in einer Kette bis zu 504 Zeichen sämtliche Vorkommen eines Zeichenmusters (Suchwort) ermittelt. Die Aufrufe lauten wie folgt:

```
Call PAT1 (X, Y, E)
```

X Zeichenkette

Y Zeichenmuster

E Feld der Dimension 8. Sei e(i,j) das i-te Bit von E(j), $0 \leq j \leq 7$, $0 \leq i \leq 63$. Dann gilt: e(i,j) = 1 genau dann, wenn Y in X ab Position 8•i+j beginnt (Ausgabeparameter).

Wird das Programm unter Beibehaltung des Suchwortes mehrfach aufgerufen, so kann man nach dem ersten Aufruf von PAT1 auf eine erneute Initiierung des Algorithmus verzichten, sofern die Länge der Zeichenkette X unverändert bleibt, und schreibt:

```
Call PAT2 (X, E).
```

Das unten angegebene CAL-Programm VECMZ dient zur Erzeugung von Vektormasken, das Programm SHIFT1 verschiebt den Inhalt eines Vektorregisters bzw. eines Feldes der Dimension 64 um 8 Bits, und CHAVEC schiebt eine Zeichenkette auf Wortgrenze.

Literaturverzeichnis

1. A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search. Comm. ACM 18 (1975), 333 - 340.

2. D.E. Knuth, J.H. Harris, K.R. Pratt, Fast pattern matching in strings, SIAM J. Comp. 6 (1977), 323 - 350.

3. R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (1977), 762 - 772.

4. J. Krieger, Sortieren auf der CYBER 205. 4. Bochumer Kolloquium über Größtrechner und Anwendungen, Dezember 1983.

```
C     +-------------------------------------------------------------------
C     : TESTPROGRAMM FUER PAT1
C     +-------------------------------------------------------------------
      PROGRAM PATEST
      LOGICAL E(0:7)
      CHARACTER C*480,B*22
      DO 17 I=1,401,80
         READ (*,   '(A80)') C(I:I+79)
   17    WRITE(*,'(1X,A80)') C(I:I+79)
      READ (5,'(A22)') B
      DO 21 I=1,22
         WRITE(*,*) B(I:)
         CALL PAT1(C,B(I:),E)
         K=0
         DO 20 J=0,63
            DO 19 L=0,7
               K=K+1
               IF (E(L)) WRITE(*,*) K
   19          E(L)=SHIFTL(E(L),1)
   20       CONTINUE
   21    CONTINUE
      END
```

```
      SUBROUTINE PAT1(X,Y,E)
C     +-------------------------------------------------------------
C     : PARAMETER  :  ZEICHENKETTE X; JEDE LAENGE BIS 504 (CRAY GRENZE)
C     :               ZEICHENKETTE Y; JEDE LAENGE BIS 504 (CRAY GRENZE)
C     :               INTEGER (BOOLEAN) FELD E(0:7)
C     : AUFGABE    :  JEDES VORKOMMEN VON Y IN X SOLL GEFUNDEN WERDEN.
C     :               IST IN E(J) DAS I-TE BIT 1 (I=0,1,...63),
C     :               DANN KOMMT Y IN X AN DER STELLE I*8+J+1 VOR.
C     : COMPILATION:  OPT=BTREG:FASTMD
C     +-------------------------------------------------------------
      IMPLICIT INTEGER(A-Z)
      CHARACTER*(*) X,Y
      DIMENSION E(0:7),A(0:63),B(0:63),C(0:63)
C
      LX=LEN(X)
      LY=LEN(Y)
      NWXP1=LX/8+1
      NWY=LY/8
      RESTY=LY-NWY*8
      CALL CHAVEC(Y,B)
      MAS=MASK(RESTY*8)
      BREST=AND(MAS,B(NWY))
      NPRE=LX-LY+512
C
      E N T R Y   PAT2(X,E)
      CALL CHAVEC(X,A)
C     +-------------------------------------------------------------
C     : X (HIER A) WIRD UM 0 BIS 7 ZEICHEN NACH LINKS VERSCHOBEN.
C     +-------------------------------------------------------------
      DO 5, ISHI=0,7
         E(ISHI)=MASK(64)
         NVERGL=(NPRE-ISHI)/8-64
C        +----------------------------------------------------------
C        : SUCHE IN X DIE VORDEREN 8-ZEICHEN-ANTEILE (WORTE) VON Y.
C        +----------------------------------------------------------
         DO 2, IWORT=0,NWY-1
CDIR$       SHORTLOOP
            DO 1, I=IWORT,IWORT+NVERGL
   1           C(I)=A(I)-B(IWORT)
            F=0
            IF (NVERGL.GE.0) F=VECMZ(NVERGL+1,C(IWORT),1)
            E(ISHI)=AND(E(ISHI),F)
            IF (E(ISHI).EQ.0) GOTO 4
   2        CONTINUE
C        +----------------------------------------------------------
C        : SUCHE IN X NACH DEM LETZTEN TEILWORT VON Y (LAENGE < 8).
C        +----------------------------------------------------------
         IF (RESTY.NE.0) THEN
CDIR$       SHORTLOOP
            DO 3, I=NWY,NWY+NVERGL
               C(I)=AND(MAS,A(I))
   3           C(I)=C(I)-BREST
            F=0
            IF (NVERGL.GE.0) F=VECMZ(NVERGL+1,C(NWY),1)
            E(ISHI)=AND(E(ISHI),F)
            END IF
   4     IF (ISHI.LT.7) CALL SHIFT1(NWXP1,A)
   5     CONTINUE
      END
```

```
         IDENT     CHAVEC          BEGIN OF COMMENT
*                                  DIE SUBROUTINE CHAVEC HAT 2 PARAMETER
*                                  DER 1. IST EINE ZEICHENKETTE CHA MIT
*                                      BELIEBIGEM ANFANG INNERHALB EINES
*                                      WORTES.
*                                  DER 2. EIN STARTFELDELEMENT VEC.
*                                  FUNKTION: DIE ZEICHENKETTE CHA WIRD
*                                   LINKSBUENDIG AUF DAS FELD VEC
*                                   UMGESPEICHERT. VEC HAT INCREMENT 1.
CHA      DEFARG
VEC      DEFARG
CHAVEC   ENTER     MODE=BASELVL,PRELOAD=0
         ARGADD    S1,CHA,ARGPTR=A6
         ARGADD    A2,VEC,ARGPTR=A6
         A0        S1              WORTADRESSE VON CHA
         S3        <6
         S2        S1
         S2        S2>24
         S4        S3&S2           OFFSET IN BIT
         A4        S4
         S2        S2>6            CHAR-LAENGE IN BIT
         S5        S4+S2
         S5        S5>6            VEKTORLAENGE-1
         A3        S5
         A5        A3+1            VEKTORLAENGE
         VL        A5
         V1        ,A0,1           LADE VEKTOR MIT CHARACTERN
         A0        A2
         V2        V1,V1<A4        SCHIEBE UM A4 BITS (UM DEN OFFSET)
         ,A0,1     V2
         EXIT      MODE=BASELVL
         END
```

```
         IDENT     VECMZ          BEGIN OF COMMENT
*                                 DIE FUNCTION VECMZ  HAT 3 PARAMETER.
*                                 DER 1. IST DIE FELDLAENGE LENVEC:
*                                              0<=LENVEC<=64.
*                                 DER 2. DAS STARTFELDELEMENT MEMVEC.
*                                 DER 3. IST DAS FELDINCREMENT INCVEC.
*                                 RETURN: IN VECMZ IST BIT I EINE 1,WENN
*                                 MEMVEC(I)=0;SONST 0(I=1,...,LENVEC).
*                                 FUER I=LENVEC+1, ... , 64 : BIT I=0.
LENVEC   DEFARG
MEMVEC   DEFARG
INCVEC   DEFARG
VECMZ    ENTER     MODE=BASELVL,PRELOAD=0
         ARGADD    A7,LENVEC,ARGPTR=A6      A7=ADRESSE VON LENVEC.
         ARGADD    A2,MEMVEC,ARGPTR=A6      A2=ADRESSE 1. FELDELEMENT
         ARGADD    A3,INCVEC,ARGPTR=A6      A3=ADRESSE DES INCREMENTS.
         A4        ,A7              A4=LENVEC
         A5        ,A3              A5=INCVEC
         VL        A4               VL=LENVEC
         A0        A2               A0=STARTADRESSE DES FELDES
         V1        ,A0,A5           V1=VEKTOR:START A0,INCR A5,LEN VL
         VM        V1,Z             VECTOR MASK WIRD 1 FUER V1=0
         S1        VM
         EXIT      MODE=BASELVL
         END
```

```
         IDENT     SHIFT1          BEGIN OF COMMENT
*                                  SHIFT1 HAT ZWEI PARAMETER.
*                                  PARAMETER VLEN IST DIE FELDLAENGE.
*                                  PARAMETER VCHAR IST EIN FELD, INHALT
*                                  VOM TYP CHARACTER (AUF WORTGRENZE).
*                                  SHIFT1: VERSCHIEBE VCHAR UM 8 BIT
*                                          (1 BYTE) NACH LINKS.
VLEN     DEFARG
VCHAR    DEFARG
SHIFT1   ENTER     MODE=BASELVL,PRELOAD=0
         ARGADD    A7,VLEN,ARGPTR=A6       A7=ADRESSE VON VLEN
         ARGADD    A2,VCHAR,ARGPTR=A6      A2=ADRESSE VON VCHAR
         A7        ,A7             A7=VLEN
         VL        A7              VL=VLEN
         A3        8               A3=8
         A0        A2              A0=STARTADRESSE
         V1        ,A0,1           V1=VEKTOR START A2 INCR 1 MIT VL ELEM
         V2        V1,V1<A3        SCHIEBE LINKS A3 BITS VL ELEMENTE
         ,A0,1     V2              STORE V2 START A0 LENGTH VL INCR 1
         EXIT
         END
```